Agent-Oriented Formal Specification of Web Services

Hong Zhu[(1)], Bin Zhou[(2)], Xinjun Mao[(2)], Lijun Shan[(2)], David Duce[(1)]

[(1)] Department of Computing, Oxford Brookes University, Oxford OX33 1HX, UK
[(2)] Dept. of Computer Science, National Univ. of Defense Tech., Changsha, 410073, China
Email: hzhu@brookes.ac.uk

Abstract. Web services (WS) provide a technology for integrating applications over the Internet. The components of a WS are active and persistent computational entities that have autonomous and social behaviours. The paper investigates the formal specification of WS architecture and applications within a caste-centric framework of multi-agent systems. An abstract specification of the general architecture of WS and an example of WS application are given in the SLABS language, which was designed for developing agent-based systems.

1 Introduction

As a distributed computing technology, Web services (WS) offer a promising approach to integrate applications over the Internet [1]. It is characterised by the dominance of program-to-program business-to-business interactions [2], hence widely recognised to be fundamentally different from existing distributed computing techniques.

The development of WS applications is bound to be complex and difficult for two main reasons. First, WS technology enables dynamic software integration at application level. Program-to-program interaction established at runtime implies that it may be impossible to determine the scope of integration at design time. There is little theory and practice of such integration in the software engineering literature. Second, business-to-business interaction implies that the integration can be within an enterprise as well as between enterprises. Thus, the software components in a WS application are usually developed by different vendors. The lack of communications between component providers and component users has long been recognised as a main cause of difficulties in component technology, but no satisfactory solution has been found. In the context of WS, recently, it is realised that, in addition to the descriptions of the syntactical aspects such as the formats of the messages, the description of semantic aspects such as business logic are of vital importance for the success of WS technology [3, 4]. Proposed solutions in the literature rely on ontologies for taxonomic descriptions of the functionality of each service, and on workflow for the restrictions on the orders that services are called [5, 6]. It is still unclear whether ontology and workflow descriptions are adequate to provide the required semantic information.

In this paper, we propose an approach that uses formal specifications to describe the semantic aspects of WS based on our caste-centric framework of multi-agent systems (MAS) and illustrate the uses of an agent-oriented formal specification language SLABS [7, 8] to bridge the gulf between service providers and requesters.

2 Web Services as MAS

Agency is a fundamental concept in agent-based computing though what agenthood means exactly is a matter of controversy. People tend define the concept by certain characteristic properties [9, 10]. Among many such properties, autonomy, pro-activity, responsiveness and social ability have been widely considered as the most important. These properties match the features of software systems that constitute a WS application. The components of a WS application can be considered as software agents. For example, each provider or requester is autonomous. It can say 'go' to initiate actions such as to request for services. It can also say 'no' so as to refuse a service request. These components have certain social ability because of their dynamic discovery and invocation of services. At this level of abstraction, it is apparent that agent technology is suitable for the development of WS applications.

However, not all agent models are suitable for the development of WS. For example, BDI models define agents as computational entities that have mental states that consist of belief, desire and intension [11, 12]. In such models, agents' behaviours are controlled by such mental states. Game theory models define agents as computational entities that aim to maximise their utility functions. WS has been considered as an attractive technology for wrapping existing applications and IT assets so that new solutions can be deployed quickly and recomposed to address new opportunities [2]. Few of existing IT assets can be considered as agents in these models.

Therefore, this paper take a software engineering approach to the analysis, modelling and design of MAS [13]. We define agents as active and persistent computational entities that encapsulate data, operations and behaviours and situate in their designated environments. Here, data represents an agent's state. Operations are the actions that an agent can take. Behaviours are rules that govern the agent's state changes and actions. By encapsulation, we mean that an agent's state can only be changed by the agent itself. In our model, agents' structure consists of a name, an environment description, a list of state space and action declarations, and a body in the form of Fig. 1 that determines its behaviour.

```
Begin
    Initialise state;
    Loop
        Perceive the visible actions and states of the agents in
            its environment;
        Take actions and change state according to the situa-
            tion in the environment and its internal state;
    end of loop;
end
```

Fig. 1. The control structure of agent's body

The central concept of our approach is *caste*, which is the classifier of agents. It is a new concept introduced by SLABS. In our model, the agents in a MAS are grouped into castes. The agents in the same caste have a set of common structural and behavioural characteristics. An example of behaviour characteristics is that an agent follows a specific communication protocol to communicate with other agents. The relationship between agents and castes is similar to that between objects and classes. The difference is that an agent can join a caste and retreat from a caste dynamically at run-time. Inheritance relationships can also be defined between castes. A sub-caste inherits the structure and behaviour features from its super-castes. However, a sub-caste cannot override the structure and behaviour rules of a super-caste, although it can have some additional state variables, actions and behaviour rules. The parame-

ters of the super-castes may also be instantiated in a sub-caste. The caste facility provides a powerful vehicle to describe the normality of a society of agents. Multiple inheritances are allowed to enable an agent to belong to more than one society and play more than one role in the system at the same time. Castes plays a central role in our methodology of agent-oriented software development [13, 14]. It distinguishes our approach from the others. In the SLABS language, castes are specified in the form shown in Fig. 2.

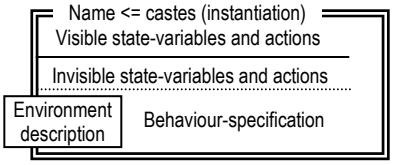

Fig. 2. SLABS's specification of castes

The components of a WS application can be modelled as agents defined above. They are divided into castes of service providers and service requesters. Different types of service requesters can also be further grouped into sub-castes so that components representing different types of service requesters are divided into the different sub-castes and have different structural and behavioural features. An agent can join a sub-caste to become a valid requester and retreats from the caste after the service is finished or when it is unsatisfied with the service. When it is a member of the caste, it must obey the behaviour rules in order to obtain the required services. But, it has no obligations to follow the rules after it retreats from the caste.

Agents are situated in their *designated* environments. By designated environment, we mean that the environment of an agent contains a specified subset of the entities in the system. This subset may vary at run-time within a specified range. In SLABS, a environment description specifies a collection of castes and a set of particular agents. A designated environment differs from a completely *open environment*, where every element in the system can always affect the behaviour of an agent. It also differs from a *fixed environment*, where an agent can only be affected by a fix set of entities in the environment. In both fixed and open environments, the agent cannot change its environment. It is worth noting that both fixed and open environments are special cases of the designated environments.

3 Specification of WS Architecture

The architecture of WS covers three main aspects of distributed computing: (a) a framework of the organisation of the software systems for access through a network; (b) the mechanism and facility for the publication and registration of the services so that the services can be dynamically discovered; (c) a set of standards that enables components to exchange data with each other. In particular, the provided services are described in WSDL using a standard formal XML notation that provides all of the details necessary to interact with the service including message format, transport protocol and location. The services are published with a service registry that complies with a standard called UDDI. Once a WS is published, a service requester can find the service via the UDDI interface. Standards like HTTP, SOAP and XML are used for transportation and marshalling of parameters so that platform and language-independent access to WS can be achieved.

At an abstraction level above the technical details, the architecture of WS consists

of three types of components: the service registry, the service providers, and service requesters. These agents of belong to three different castes specified below.

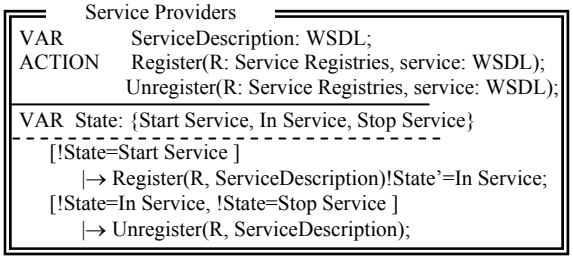

Fig. 3. Specification of service provider

The caste in Fig. 3 specifies service providers. It states that a service provider can have two actions: to register and unregister at a service registry. It has a visible state that describes its services. Its behaviour is specified by two rules: one for register and the other for unregister.

The caste in Fig. 4 specifies service requesters. A service requester can make search requests to a service registry, but there is no restriction on when and what to search for. Therefore, there is no behaviour rule in the body of the caste.

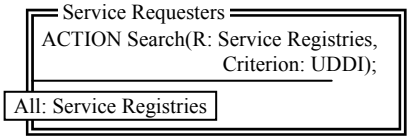

Fig. 4. Specification of service requester

Fig. 5 is the specification of service registries in SLABS. There are three rules for the behaviour of a service registry. The first states that when a service requester searches for a WS with a criterion, the registry must reply with a set of registered WS that matches the criterion. Here, we leave the function *Match* as a predefined function. The second and third rules deal with registration and unregistration, respectively.

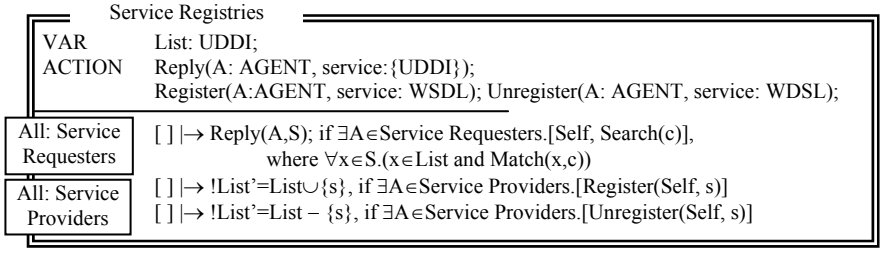

Fig. 5. Specification of service registries

Notice that, first, the semantics of SLABS implies that an agent can be a member of one or more castes. For example, a service provider can also be a service requester of another service provider. Second, an agent can join a caste and retreat from a caste at run-time. The membership relation is not static. Third, in the specification above, instead of giving all the details of the standards UDDI, SOAP and WSDL, we treat them as pre-defined data types and provide an abstract specification of the functionality and behaviour of the components. This enables us to focus on the logic of WS rather than syntactic and format details. Fourth, at the architectural level, there is no relationship between the service providers and service requesters. The interactions between them can be established at runtime and specified with the particular service provider and requester. Finally, the specifications given in this paper are for the illustration of the uses of SLABS. Some simplifications of the problems are made.

4 Specification of WS Service Providers

The specification of a service provider not only needs to define the services that it provides, but also the way that the services should be used. In a WS application, service requesters can be further classified into a number of types. Each of them can be specified by a caste.

For example, consider the online auction services. Two types of requesters may interact with an online auction WS. Sellers ask for the service provider to set up an online auction to sell its goods with certain conditions. Buyers can then bid for the goods online. Thus, we identify three different castes in this application: (a) Auction Service Providers, (b) Sellers, (c) Buyers. The caste in Fig. 6 specifies the behaviour of auction providers.

```
Auction Service Provider <= Service Providers
VAR  AuctionInfo: {(ItemDetail: GOODS, Seller: Sellers;
    Current_Bid: BID, Current_Bidder: Buyers, Current_BidID: BidID
    Start, End: DATE_TIME, ID: AuctionID; Commision_Rate, Minimum_Price: REAL)};
ACTION     Accept_Auction(Sellers, AuctionID); Announce (GOODS, AuctionID);
    Accept_Member(Buyers, AuctionID, MembershipID);
    Bid_Received (Buyers, MembershipID, BID); Bid_Accepted (Buyers, MembershipID, BID);
    Bid_Failed (Buyers, MembershipID, BID);
-------------------------------------------------
VAR  Members: {(id: AuctionID, A:Buyers, mid: MembershipID)}
ACTION     Check_Credit (Buyer):{OK, FAIL};
           Clear_Payment(Buyer, Payment);       Tranfer(Seller, Payment);
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
<Accept Auction>: [ ] |→ (Accept_Auction(A, AID) ! AuctionInfo'=AuctionInfo+(Auct));
All: Sellers     Annouce_Auction(Item_info, AID),
                      if ∃A∈Seller:[RequestAuction(Item_info, sd, ed, mp, cr)]
All: Buyers           where Auct.ItemDetail = Item_info & Auct. ID = AID
                            & Auct.Start=sd & Auct.End=ed
                            & Auct.Minimum_Price=mp & Auct.Commision_Rate=cr
<Accept Member>: [ ] |→ Accept_Member(A, AID, MID) ! Member'=Members+(A, AID, MID);
           if ∃A∈Buyers:[Join_Auction(Self, AID)]    where Check_Credit (A) = OK;
<Receive Bid>: [ ] |→ Bid_Received (A, MID, Bid_ID);
           if ∃A∈Buyers:[Submit_Bid(AID, MID, Bid)] where (A, AID, MID)∈Members;
<Failed Bid>: [Bid_Received (A, MID, Bid_ID)] |→Bid_Failed (A, MID, Bid_ID);
           if ∃A∈Buyers:[SubmitBid(AID, MID, Bid)]
          where Auct∈AuctionInfo & Auct.ID=AID & Not Beat(Bid, Auct.Current_Bid)
<Update Bid>: [Bid_Received (A, MID, Bid_ID)]
      |→ !Auct.Current_Bid'=Bid & Auct.Current_Bidder'=A & Auct.Current_BidID'=BidID);
                if ∃A∈Buyers:[SubmitBid(AID, MID, Bid)]
                where Auct∈AuctionInfo & Auct.ID=AID & Beat(Bid, Auct.Current_Bid)
<Accept Bid>: [ ]|→Auct.End:Bid_Accepted(Auct.Current_Bidder,Auct.ID, Auct.Current_BidID)
        where Auct∈AuctionInfo;
<ClearPayment>: [Bid_Accepted (A, AID, BidID)]
      |→ Clear_Payment(payment);Transfer(Auct.Seller, Deduct(Payment,Auct.cr));
           if A:[Pay(Bid_ID, AID, payment)]
           where Auct∈AuctionInfo & Auct.ID=AID& Payment_OK(Bid_ID, payment)
```

Fig. 6. Specification of auction service provider

Auction Service Providers is a sub-caste of Service Providers. Sellers and Buyers castes in Fig. 7 and Fig. 8 are sub-castes of the Service Requesters caste.

The interactions between a service provider and a requester are often so complicated that an interaction protocol must be defined. In the online auction example, the protocol defines how to bid and who will be the winner, etc. It is defined by two sets of rules, one for the auctioneer and one for the buyers. The protocol specified in Fig. 8 is a simplified version of English auction. The rules restrict the behaviour of a buyer in an auction, but not on how individuals make decisions. Similarly, a protocol for the interaction between a seller and the auction service provider must be defined and specified. Details are omitted for the sake of space.

```
Sellers <= Service Requesters
VAR  BusinessInfo:       UDDI;
ACTION     RequestAuction ( ItemInfo: GOODS,
     StartDateTime, EndDateTime: DATE_TIME,
     MinimumPrice, CommissionRate: REAL );
… …
```

Fig. 7. Specification of seller

```
Buyers <= Service Requesters
VAR  BusinessInfo: UDDI;
ACTION     Submit_Bid(AuctionID, MembershipID, BID);
        Pay(BID_ID, PAYMENT);   Join_Auction(Auction Service Providers, AuctionID);
VAR  Membership: {Yes, No};  MID: MembershipID; Auction: AuctionID; Bid_ID: BID_ID;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
<Join Auction>: [!Membership= No ] |→ time: Join_Auction(Auctioneer, AID);
    if Auctioneer:[Announce_Auction(d, AID)];
    where Auct∈ Auctioner.AuctionInfo& time < Auct.Start & Auct.ID=AID
<Get Membership ID>:
    [Join_Auction(Auctioneer, AID)]|→ !Membership'=Yes & Auction'=AID, MID'=mid
    if Auctioneer:[Accept_Member(Self, AID, mid)
<Submit Bid>:   [!Membership=Yes] |→ Submit_Bid(Auction, MID, Bid);
    where Beat(Bid, Auctioneer.auct.Current_Bid) & Auct∈ Auctioneer.AuctionInfo
    & Auction.Auct.ID=Auction
<Receive Acknowledge Of Bid>: [Submit_Bid(Auction, MID, Bid)] |→!Bid_ID'=bidID;
    if Auctioneer:[ Bid_Received (Self, AID, mid, bidID)], where AID=Auction & mid = MID;
<Revise Bid After Failure>: [Submit_Bid(Auction,MID,Bid)] |→; Submit_Bid(Auction,MID, Bid2)
    If Auctioneer:[Bid_Failed(Self, AID, mid, bidID), $^k],
    where Auct ∈ Auctioneer.AuctionInfo & Auct.ID=Auction
           & Beat(Bid, Auct.Current_Bid) & Bid_ID = bidID & MID=mid;
<Pay Accepted Bid>:   [Submit_Bid(Auction, MID, Bid)] |→; Pay(Bid_ID, Payment)
    If Auctioneer:[ Bid_Accepted (Self, AID, mid, bidID)],
    Where AID=Auction & Bid_ID=bidID & MID = mid
<Quit From Auction>: [!Membership=Yes] |→ Quit_Auction(AuctionID)!Membership'=No,
    if Auctioneer:[Bid_Failed(Self, AID, bidID), $^k]; where Auction=AID & Bid_ID = bidID

Auctioneer:
Auction Service
Provider
```

Fig. 8. Specification of buyer

It is worth noting that in the above example a WS service provider is specified by one caste to define the provider's functionality and behaviour together with two castes to specify the expected behaviours of the service requesters. The specification of the requesters serves as the assumptions about the requesters' actions and behaviours. It explicitly states how the services should be used. The correctness of an implementation of a WS service provider can only be understood and proved by using all of these castes. Such information is crucial for software developers not only on the service provider side but also on the service requester side. The specification of the requesters also leaves a great space of flexibility about their behaviour. For example, a specific buyer can have its own rules to determine when and what bid is to be submitted.

5 Specification of WS Service Requesters

To demonstrate how such a specification can be used for the development of requester side software, consider an online flight ticketing service that sells air tickets for an airline. Assume that, the specific application has a more concrete rule for deciding when to request online auction services. For example, the caste in Fig. 9 specifies a business rule that it will try to sell the unsold tickets by online auction when the time reaches 8 days before the scheduled flight.

The caste SellByAuction in Fig. 9 inherits the capability and behaviour of the caste Sellers for its interaction with auction service providers and a caste TicketSellers for its business rules. It also has an additional rule for its request of auction services. In general, the specification of business logic can be separated from the specification of the interaction protocol by using two or more castes.

```
SellByAuction <= TicketSellers, Sellers
-----------------------------------------
[ ]|→ Flight.Date−8: (* 8 days before the departure date. *)
    RequestAuction(
        Auctioneer, (* the auction service provider *)
        <AirTicket, Flight.No, Seat>, (* product information *)
        Flight.Date-7, (* Start date of auction *)
        Flight.Date-1, (* End date of auction *)
        Flight.MinPrice, (* Minimum price *)
        10% ), (* Commission rate *)
    Where Flight ∈ AirTicketing.ListOfFlights
        & Flight.MaxSeats > Flight.SoldSeats
        & Seat ∈{1, 2, …, Flight.MaxSeats – Flight.SoldSeats}

Auctioneer: Auction Service Providers
```

Fig. 9. Specification of air ticket seller who sells by auctions

An auction service requester may use a number of different auction service providers, say auctioneer A and B, to sell their products such as air tickets. In such a case, we can declare two agents as instances of the caste SellByAuction. Alternatively, agent A and B can be dynamically created as instances of the caste. Details of their specifications are omitted for the sake of space.

4 Concluding remarks

The approach to the formal specification of WS proposed in this paper can be summarised by two well-known software engineering principles. The first is the *principle of separation of concerns*. The specification of different kinds of components such as the providers and requester are separated into different castes. Different types of WS requesters and providers are further separated into sub-castes. The specification of private information such as business logic and internal decision making processes are separated from the specification of public information such as interaction protocols, communication protocols, etc. and specified in different castes. Such a modular structure of specification enables the application of the second principle, which is the *principle of information hiding*. The private information isolated in a caste can be hidden from public access. At the same time, the public information, especially the assumptions made by the service provider about the service requesters are specified. These principles are strongly supported by the caste facility. The specifications in SLABS are modular, composable and reusable.

There have been several efforts to define specification languages and/or standards

for enabling software to use WS. Among the most well-known are IBM's WSFL [5] based on Petri Net theory, Microsoft's XLANG [6] rejuvenated the Pi-Calculus model, and *BPMI.org*'s BPML 1.0 [15] that unified these two approaches. More recently, BEA, IBM, and Microsoft published BPEL4WS. Other organizations advocated radically different approaches for business process modeling, such as DAML-S [16]. There are two most important differences between SLABS and the above. First, WSFL, BPML and DAML-S focus on the workflow management of multiple Web Services, i.e. the execution orders and transactional issues. SLABS can specify these issues as well as other semantic aspects of Web Service. Second, SLABS is on a more abstract level while the related works are on a more operational level.

There are a number of problems that need further research. We are investigating how formal specifications of WS can be represented in XML format and facilitate the dynamic search and integration of WS applications.

Acknowledgement. The work reported in this paper is partly supported by China National High Technology Research and Development Programme (863 programme) under the grant 2002AA116070.

References

[1] Lau, C. and Ryman, A., Developing XML Web services with WebSphere Studio Application Developer. IBM SYSTEMS JOURNAL, 2002. 41(2): pp178-197.
[2] Gottschalk, K., et al., Introduction to Web services architecture. IBM SYSTEMS JOURNAL, 2002. 41(2): pp170-177.
[3] Leymann, F., Roller, D., and Schmidt, M.-T., Web services and business process management. IBM SYSTEMS JOURNAL, 2002. 41(2): pp198-211.
[4] Lambros, P., Schmidt, M.-T., and Zentner, C., Combine Business Process Management Technology and Business Services to Implement Complex Web Services, IBM Corp, 2001.
[5] Leymann, F., Web Services Flow Language, IBM Corporation, 2001.
[6] Thatte, S., XLANG-Web Services for Business Process Design, Microsoft Corp., 2001.
[7] Zhu, H., SLABS: A Formal Specification Language for Agent-Based Systems. Int. J. of Software Engineering and Knowledge Engineering, 2001. 11(5): pp529-558.
[8] Zhu, H., A Formal Specification Language for Agent-Oriented Software Engineering, Department of Computing, Oxford Brookes University, 2002.
[9] Jennings, N.R., On agent-based software engineering. Artificial Intelligence, 2000. 117: pp277-296.
[10] Lange, D.B. Mobile Objects and mobile agents: The future of distributed computing? in Proc. of Proceedings of The European Conference on Object-Oriented Programming, 1998.
[11] Rao, A.S. and Georgreff, M.P. Modeling Rational Agents within A BDI-Architecture. in Proc. of the Int. Conf. on Principles of Knowledge Rep. and Reasoning, 1991, pp473~484.
[12] Wooldrighe, M., Reasoning About Rational Agents, 2000: The MIT Press.
[13] Shan, L. and Zhu, H. CAMLE: A Caste-Centric Agent-Oriented Modelling Language and Environment. in Proc. of SELMAS'04 at ICSE'94, Edinburgh, UK., 2004, IEE, pp66-73.
[14] Zhu, H., The role of caste in formal specification of MAS, in Proc. of PRIMA'2001. LNCS, Vol. 2132, 2001: Springer: Taipei, Taiwan, pp1~15.
[15] BPML.org, The BPML specification version 1.0, http://www.bpmi.org.
[16] Daml.org, The DAML Services Coalition. DAML-S: A Semantic Markup For Web Services, http://www.daml.org/services/daml-s/2001/10/daml-s.pdf.